universität innsbruck

Functional Programming	WS 2020	LVA 703025
Exercise Sheet 5, 10 points		Deadline: Wednesday, November 11, 2020, 6am

- Please write all the Haskell code into a single .hs-file and upload it in OLAT.
- You can use the template .hs-file that is provided on the proseminar page.
- Your .hs-file should be compilable with ghci.
- Don't forget to mark your completed exercises in OLAT.

Exercise 5.1 *Functional decomposition, Recursion, Character Strings* **4 p.**

This exercise is about manipulating character strings (type `String` in Haskell). To solve the following subitems, the only built-in functions you may use are `++` (to append two strings), `null` (to check whether a string is empty), `head` (to obtain the first character of a non-empty string), and `tail` (to obtain everything except for the first character of a non-empty string).

1. Implement a function `samePrefix :: String -> String` that, given an input string, returns its longest prefix consisting of the same character. For example:

   ```
   samePrefix "" == ""
   samePrefix "abc" == "a"
   samePrefix "aaah!" == "aaa"
   ```

   Hint: Use `[c]` to turn a single character `c` into a string. (1 point)

2. Implement a function `dropSamePrefix :: String -> String` that, given a string, returns the result of removing the longest prefix consisting of the same character. A correct implementation should satisfy the equation `samePrefix s ++ dropSamePrefix s == s` for arbitrary strings `s`. (1 point)

3. Implement a function `reverseString :: String -> String` that reverses a string, using `samePrefix` and `dropSamePrefix` from above. For example:
   `reverseString "Hello World!" == "!dlroW olleH"` (2 points)

Exercise 5.2 *Arbitrary Precision Natural Numbers* **6 p.**

This exercise is about implementing arbitrary precision (non-negative) integers using strings. We represent numbers by strings of digits in reverse order, for example, 10 is represented by `"01"`, 199 by `"991"`, etc.
To solve the following subitems you may use `:`, `++`, `null`, `head`, and `tail`. Here, `:` is a binary operation of type `Char -> String -> String` that adds a character in front of a string, and the `Char`-type can be seen as an enumeration-type with constructors such as `'a'`, `'b'`, `'A'`, `'0'`, `'1'`, ... For instance, `'6' : "234" = "6234"`.

1. Implement two functions `fromInteger :: Integer -> String` and `toInteger :: String -> Integer` that translate between Haskell integers and our String representation (for the sake of functional decomposition you should first implement `fromDigit :: Integer -> Char` and `toDigit :: Char -> Integer` that translate between digits and characters). For example:

   ```
   fromInteger 19 == "91"
   toInteger "24" == 42
   ```

   (2 points)

   In the remainder you are not allowed to use `fromInteger` and `toInteger` to perform computations on integers and then translate into strings. Instead you should work directly on strings.

2. Implement a function `add :: String -> String -> String` that performs addition on arbitrary precision numbers (build on top of `addDigits :: Char -> Char -> String`, adding two digits; you are only allowed to use `+ :: Integer -> Integer -> Integer` on digits). For example:
`"99" `add` "104" == "005"` (2 points)

3. Implement a function `pred :: String -> String` that computes the predecessor of an arbitrary precision number (that is, subtract 1 from the number, but do not go below 0; you are only allowed to use `- :: Integer -> Integer -> Integer` on digits). For example:
`pred "0" == "0"`
`pred "001" == "99"` (2 points)

Hint: Start your Haskell script by `import Prelude hiding (fromInteger, toInteger, pred)` to hide the built-in functions that share names with the functions in this exercise.